1 SUPPLEMENTARY INFORMATION

In this section, we derive the upper bound on the size of the data structures used to organize the database index and describe the "seed search algorithm" used to find initial identical substrings in the query and the database. The seed search algorithm has been incorporated into a modified version of NCBI's MegaBLAST, as described below, with few changes to the algorithmic code parts that do subsequent processing of the seeds.

1.1 Index Size Upper Bound

In this subsection, the size of an index volume in bytes is estimated as a function of the size of the sequence database, k, and s. The size estimate is useful to understand when our implementation of database indexing is practical. Let V be the number of input sequences contributing to the volume, and n be the total number of bases in the input sequences. In this subsection, we use *word* to refer to a 4-byte unsigned integer value.

An index volume stores the length of each input sequence, which takes one word per sequence, contributing $4V$ bytes to the size. The mapping of logical sequence ids to constituent input sequence ids takes 4 words per logical sequence, contributing $4n/2^{B-1}$ bytes. The denominator comes from the fact that concatenation of real sequences into logical sequences ensures that pairs of consecutive logical sequences, except possibly the last, have length at least $2^B + 1$.

The sequence data are stored in the sequence store using 2 bits per base encoding. At most 3 bases worth of padding is added after each sequence to align them to a byte boundary. Consequently, the space taken by the sequence store can be bounded by $\lfloor n/4 \rfloor + \frac{3}{4}V$ bytes.

The lookup table contains 4^k words and is terminated by a 0-valued word. Each offset list is terminated by a 0-valued word. So in the worst case, where every k-mer has a non-empty offset list, the lookup table together with 0-value terminating words occupy $4 + (2 \times 4^{k+1})$ bytes.

The size of the space occupied by the offset lists depends on the number and lengths of the seed-eligible intervals. Let M be the total number of valid intervals and $m_i,\ 1 \le i \le M$ be the length of the i-th seed-eligible interval. Let $m = \sum_{i=1}^{M} m_i$ be the combined length of all valid intervals. i-th seed-eligible interval has at most $\lfloor (m_i - (k-1))/s \rfloor$ entries in the offset lists, and at most 2 of them are special offsets. Hence, it contributes at most $\lfloor (m_i - (k-1))/s \rfloor + 2$ words to the total index size. The total size of the offset lists is then estimated from above by

$$\sum_{i=1}^{M} \lfloor m_i/s + (2 - (k-1)/s) \rfloor \tag{1}$$

We consider two cases.

If $k \ge 2s + 1$, then $2 - (k-1)/s < 0$ and expression (1) does not exceed $\lfloor m/s \rfloor \le n/s$.

If $k < 2s + 1$ or, equivalently, $s > (k-1)/2$, then expression (1) does not exceed

$$\sum_{i=1}^{M} \left(2 + \frac{m_i - (k-1)}{(k-1)/2}\right) = \sum_{i=1}^{M} \frac{m_i}{(k-1)/2}$$
$$= \frac{m}{(k-1)/2} \le \frac{n}{(k-1)/2}$$

Combining the two cases we get that the total size occupied by the offset lists is at most

$$\frac{n}{\min\left(s, \frac{k-1}{2}\right)}$$

words.

For real biological sequence databases, one can assume that the number of input sequences V and the total number of logical sequences are both much smaller than n. Under this assumption, the total size of the index can be estimated as approximately

$$2 \times 4^{k+1} + \frac{n}{4} + \frac{4n}{\min\left(s, \frac{k-1}{2}\right)} \tag{2}$$

bytes. For example for an unmasked database of size 1 Gb and our default values $k = 12, s = 5$, the expected size of the index would be approximately 1.175 Gb.

1.2 Search Procedure

The seed search algorithm identifies longest exact matches (seeds) between the query and the database that are at least wsize $\ge w$ bases long, where wsize is the parameter of the procedure and represents the minimum target seed length. Both a database and a query may contain ambiguous or masked bases. The seeds are required to contain neither masked nor ambiguous parts of the query or the database.

If (s, q) is a pair of matched base pair positions in the database and the query respectively, then their *diagonal* $\mathrm{diag}(s, q)$ is defined as $s - q$.

At any point during its execution, the procedure maintains a set of seed candidates, which is implemented as an array indexed by logical sequence ids. Each element of the array is a linked list containing seed candidates for the corresponding logical sequence. Seed candidates within each list are kept in increasing order by their diagonal. The algorithm described below ensures that each list has at most one candidate per diagonal so there are no ties.

The procedure works by scanning the query from left to right identifying the k-mers in the query free from ambiguous or masked bases. For each such k-mer a corresponding offset list is pulled from the offset data section of the index. Offsets from the offset list are used to update the current seed candidate lists. During this process any seed candidates that cannot be extended further are identified and removed from the set of seed candidates. The ones that reached the target seed length of wsize are saved in the final result set.

The final result set has a similar structure to the set of seed candidates except that the results are grouped by input sequence ids rather than logical sequence ids.

The following subsections provide a simplified version of the search procedure. The production implementation contains some optimizations aimed at improving cache performance of the code and briefly described in subsection 1.2.3.

1.2.1 Outer Loop The following pseudocode shows the structure of the outer loop of the search procedure.

The search is performed by procedure FIND_SEEDS which takes a query, an index, and a target seed length as parameters. For every consecutive query position the corresponding offset list is retrieved from the offset data section of the index. Then each element of the

offset list is processed by procedure UPDATE_SEED_CANDIDATES which is described in the following section.

It is not guaranteed that the set of list candidates is empty after the whole query is processed. For this reason there is an additional loop in the end of FIND_SEEDS procedure that iterates over all seed candidate lists, checks each remaining seed candidate for length, and appends it to the corresponding result set.

We do not show pseudocode for the following utility functions but describe them with text: GET_KMER, GET_OFFSET_LIST, and SCL_CLEAR.

Function GET_KMER returns the k-mer value ending at a certain position in the sequence that can be used as key to the lookup table. Function GET_OFFSET_LIST returns an offset list corresponding to the given k-mer. Procedure SCL_CLEAR takes a seed candidate list as a parameter and removes all its elements.

Function DECODE_OFFSET takes an encoded position value from an offset list and breaks it down into the logical sequence id and the actual offset. For special positions it also returns the pair $(\mathrm{l\_extent}, \mathrm{r\_extent})$ and the flag indicating that the returned value represents a special offset. l_extent (r_extent) is the distance from the left (right) end of the k-mer that ends in the given sequence position to the left (right) end of the corresponding seed-eligible interval incremented by 1, or 0 if that distance is greater than $s - 1$.

Note. In the pseudocode below all functions and procedures take parameters by reference, $\mathrm{kmer\_size} = k$, $\mathrm{special\_stride} = s_2$, and $\mathrm{off\_bits} = B$.

Algorithm S1. Outer loop of the seed search procedure.

```
function DECODE_OFFSET(list, list_item)
    l_extent ← r_extend ← 0
    if list_item < special_stride^2 then // special offset
        special ← true
        l_extent ← list_item÷special_stride
        r_extent ← list_item mod special_stride
        // get the second entry for special offsets
        list_item ← LIST_NEXT(list, list_item)
    else
        special ← false
    end if
    divisor ← 2^off_bits
    lid = list_item÷divisor
    off = list_item mod divisor
    return (special, lid, off, l_extent, r_extent)
end function
procedure FIND_SEEDS(query, index, wsize)
    ql ← SEQUENCE_LENGTH(query)
    for qoff ← kmer_size − 1 to ql − 1 do
        kmer ← GET_KMER(query, kmer_size, qoff)
        ol ← GET_OFFSET_LIST(index, kmer)
        for item in ol do
            (special, lid, soff, l_extent, r_extent)
            ← DECODE_OFFSET(index, ol, item)
            UPDATE_SEED_CANDIDATES(index, query, wsize,
lid, soff, qoff, l_extent, r_extent)
        end for
    end for
    for each lid do // process the remaining seed candidates
        scl ← INDEX_GET_SEED_CADIDATES(index, lid)
        for each (soff, qoff, len) in scl do
            CHECK_AND_SAVE_SEED(index, lid, wsize, soff,
qoff, len)
        end for
        SCL_CLEAR(scl)
    end for
end procedure
```

1.2.2 Inner Loop The inner loop of the search procedure is implemented in the function UPDATE_SEED_CANDIDATES. The function operates on the seed candidate list for the given logical sequence. The parameters to the function include the index, the query, and the information about the new k-mer match position including its logical sequence id, subject and query offsets soff and qoff, and the information l_extent and r_extent about its distance from the boundaries of the corresponding seed-eligible interval.

Each seed candidate contains the positions of the last bases of the seed in the subject and query sequences and the seed length. Each seed candidate is also extended as far as possible to the left and to the right possibly crossing the seed-eligible interval boundary. The algorithm however ensures that when a seed is saved, it is contained within a seed-eligible interval. For each diagonal value, there can be at most one seed candidate in the list on that diagonal at any given time.

For the purposes of pseudocode below, seed candidate lists are opaque objects that internally maintain a set of seed candidates, reference to the current position, and the position qoff of the last base of the current query k-mer. Seed candidates are ordered by increasing diagonal and are operated upon via the following functions.

SCL_GET_CURRENT takes a list and a query position as parameters. If the query position is different from the qoff maintained by the list, then the internal current position in the list is reset to the beginning of the list. It returns the seed candidate at the current position.

SCL_SET_CURRENT updates the contents of the current element in the given list of seed candidates.

SCL_END returns a boolean value indicating whether the end of the list has been reached.

SCL_REMOVE removes the current seed candidate, moves the current position to the next element, and returns its data.

SCL_NEXT moves the current position to the next element and returns its data.

SCL_INSERT inserts new seed candidate data at the current position.

Let d be the diagonal of the pair $(\mathrm{soff}, \mathrm{qoff})$. Function UPDATE_SEED_CANDIDATES first considers all seed candidates with diagonals less than or equal to d. If the query position of a seed candidate is less than qoff, then first, it is considered by CHECK_AND_SAVE_SEED where it is saved if its length is $\geq$ wsize, and second, it is removed from the list.

If there is a seed candidate in the list with diagonal equal to d, and it is extended further to the right than qoff, then the function returns, since the current k-mer match on diagonal d is already covered by that seed candidate. However, if r_extent is positive, then the right end of the seed candidate is adjusted to not exceed $\mathrm{soff} + \mathrm{r\_extent} - 1$. This puts it definitely within the corresponding seed-eligible interval.

The function then tries to extend the k-mer match to the left and to the right using the sequence data from the index sequence store.

The function EXTEND_LEFT (EXTEND_RIGHT) extends by at most l_extent − 1 (r_extent − 1) bases to the left (right) or as much as possible if l_extent = 0 (r_extent = 0). It then verifies that the length of the extended match is at least wsize and inserts it into the seed candidate list at the current position.

The function, however, will insert matches that are shorter than wsize into the list if l_extent is positive. This is necessary to prevent further matches from being extended to the left beyond the left boundary of the seed-eligible interval.

Algorithm S2. Inner loop of the seed search procedure.

```
procedure UPDATE_SEED_CANDIDATES(index, query, wsize, lid, soff, qoff, l_extent, r_extent)
    len ← INDEX_GET_KMER_LENGTH(index)
    scl ← INDEX_GET_SEED_CADIDATES(index, lid)
    subject ← INDEX_GET_SEQUENCE(index, lid)
    diag ← soff − qoff
    (soff_p, qoff_p, len_p) ← SCL_GET_CURRENT(scl, qoff)
    while DIAG(soff_p, qoff_p) < diag and (not SCL_END(scl)) do
        if qoff_p < qoff then
            CHECK_AND_SAVE_SEED(index, lid, wsize, soff_p, qoff_p, len_p )
            (soff_p, qoff_p, len_p) ← SCL_REMOVE(scl)
        else
            (soff_p, qoff_p, len_p) ← SCL_NEXT(scl)
        end if
    end while
    if DIAG(soff_p, qoff_p) = diag then
        if qoff_p ≥ qoff then
            if r_extent > 0 then
                // check for crossing the boundary of seed-eligible interval
                if qoff_p ≥ qoff + r_extent then
                    delta ← qoff_p − qoff − r_extent + 1
                    qoff_p ← qoff_p − delta
                    soff_p ← soff_p − delta
                    len_p ← len_p − delta
                    SCL_SET_CURRENT(scl, qoff_p, soff_p, len_p)
                end if
            end if
            SCL_NEXT(scl)
            return
        else
            CHECK_AND_SAVE_SEED(index, lid, wsize, soff_p, qoff_p, len_p )
            SCL_REMOVE(scl)
        end if
    end if
    (soff, qoff, len) ←
    EXTEND_LEFT(subject, query, soff, qoff, len, l_extent)
    (soff, qoff, len) ←
    EXTEND_RIGHT(subject, query, soff, qoff, len, r_extent)
    if l_extent > 0 or len > wsize then
        SCL_INSERT(soff, qoff, len)
    end if
end procedure
procedure CHECK_AND_SAVE_SEED(index, lid, wsize, soff, qoff, len)
    if len ≥ wsize then
        cid ← INDEX_LID2SID(index, lid, soff)
        res_lst ← INDEX_GET_RESULTS(index, cid)
        LIST_APPEND(res_lst, soff, qoff)
    end if
end procedure
```

1.2.3 Optimizations Our implementation of the indexing scheme described above includes several optimizations aimed at improving the performance of the search procedure.

One optimization uses the fact that in order to find all matches of word size $W \geq w$ it is sufficient to know the positions of every $(W-k+1)$-th k-mer in the database. So for large values of W it is not necessary to look at every s-th k-mer. Our implementation takes advantage of this by storing the offset lists in a special order. Let, as before, p be a position of a k-mer within its logical sequence. Let t be the largest positive integer, such that $ts \leq W-k+1$. For each k-mer value we divide the set of its occurrences in the database in the collection of sets $S_i, i \in 1 \ldots t$. The sets are defined recursively as follows. The set S_t contains all occurrences with positions $(p = 0 \mod st)$ and all special positions. The set $S_i,\ 1 \leq i < t$ contains occurences with positions $(p = 0 \mod si)$ that are not already contained in sets $S_j,\ j > i$.

Such ordering makes it possible to terminate traversal of an offset list early. With $s = 5$ and the default MegaBLAST setting of $W = 28$ this reduces the number of memory accesses during offset list traversals by about $2/3$.

Another optimization is aimed at improving the cache memory performance of the procedure. Instead of processing each query position one at a time, the implementation first accumulates initial k-mer matches for several query positions in a special buffer. It then proceeds with extending all the accumulated matches to the target word size W. This strategy prevents frequent access switches between the offset and sequence data sections of the index.